Distributed Coordination

- Ordering events and achieving synchronization in centralized systems is easier.
  - We can use common clock and memory
- What about distributed systems?
  - No common clock or memory
  - happened-before relationship provides partial ordering
  - How to provide total ordering?

Event Ordering

- **Happened-before** relation (denoted by $\rightarrow$)
  - If $A$ and $B$ are events in the same process (assuming sequential processes), and $A$ was executed before $B$, then $A \rightarrow B$
  - If $A$ is the event of sending a message by one process and $B$ is the event of receiving that message by another process, then $A \rightarrow B$
  - If $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$
  - If two events $A$ and $B$ are not related by the $\rightarrow$ relation, then these events are executed concurrently.

Relative Time for Three Concurrent Processes

Which events are concurrent and which ones are ordered?

Exercise

Which of the following event orderings are true?

(a) $p_0 \rightarrow p_3$ : 
(b) $p_1 \rightarrow q_3$ : 
(c) $q_0 \rightarrow p_3$ : 
(d) $r_0 \rightarrow p_4$ : 
(e) $p_0 \rightarrow r_4$ :

Which of the following statements are true?

(a) $p_2$ and $q_2$ are concurrent processes.
(b) $q_1$ and $r_1$ are concurrent processes.
(c) $p_0$ and $q_3$ are concurrent processes.
(d) $r_0$ and $p_0$ are concurrent processes.
(e) $r_0$ and $p_4$ are concurrent processes.

Implementation of $\rightarrow$

- Associate a timestamp with each system event
  - Require that for every pair of events $A$ and $B$, if $A \rightarrow B$, then the timestamp of $A$ is less than the timestamp of $B$
- Within each process $P_i$, define a **logical clock**
  - The logical clock can be implemented as a simple counter that is incremented between any two successive events executed within a process
  - Logical clock is monotonically increasing
- A process advances its logical clock when it receives a message whose timestamp is greater than the current value of its logical clock
  - Assume $A$ sends a message to $B$, $LC(A)=200$, $LC(B)=195$
- If the timestamps of two events $A$ and $B$ are the same, then the events are concurrent
  - We may use the process identity numbers to break ties and to create a total ordering
Distributed Mutual Exclusion (DME)

• Assumptions
  - The system consists of \( n \) processes; each process \( P_i \) resides at a different processor
  - Each process has a critical section that requires mutual exclusion

• Requirement
  - If \( P_i \) is executing in its critical section, then no other process \( P_j \) is executing in its critical section

• We present two algorithms to ensure the mutual exclusion execution of processes in their critical sections

DME: Centralized Approach

• One of the processes in the system is chosen to coordinate the entry to the critical section
• A process that wants to enter its critical section sends a request message to the coordinator
• The coordinator decides which process can enter the critical section next, and its sends that process a reply message
• When the process receives a reply message from the coordinator, it enters its critical section
• After exiting its critical section, the process sends a release message to the coordinator and proceeds with its execution
• This scheme requires three messages per critical-section entry:
  - request
  - reply
  - release

DME: Fully Distributed Approach

• When process \( P_i \) wants to enter its critical section, it generates a new timestamp, \( TS \), and sends the message request \((P_i, TS)\) to all processes in the system
• When process \( P_j \) receives a request message, it may reply immediately or it may defer sending a reply back
• When process \( P_i \) receives a reply message from all other processes in the system, it can enter its critical section
• After exiting its critical section, the process sends reply messages to all its deferred requests

DME: Fully Distributed Approach (Cont.)

• The decision whether process \( P_j \) replies immediately to a request \((P_i, TS)\) message or defers its reply is based on three factors:
  - If \( P_j \) is in its critical section, then it defers its reply to \( P_i \)
  - If \( P_j \) does not want to enter its critical section, then it sends a reply immediately to \( P_i \)
  - If \( P_j \) wants to enter its critical section but has not yet entered it, then it compares its own request timestamp with the timestamp \( TS \)
    - If its own request timestamp is greater than \( TS \), then it sends a reply immediately to \( P_i \) (\( P_i \) asked first)
    - Otherwise, the reply is deferred

  - Example: \( P_1 \) sends a request to \( P_2 \) and \( P_3 \) (timestamp=10)
  - \( P_3 \) sends a request to \( P_1 \) and \( P_2 \) (timestamp=4)

Undesirable Consequences

• The processes need to know the identity of all other processes in the system, which makes the dynamic addition and removal of processes more complex
• If one of the processes fails, then the entire scheme collapses
  - This can be dealt with by continuously monitoring the state of all the processes in the system, and notifying all processes if a process fails

Token-Passing Approach

• Circulate a token among processes in system
  - Token is a special type of message
  - Possession of token entitles holder to enter critical section
• Processes logically organized in a ring structure
• Unidirectional ring guarantees freedom from starvation
• Two types of failures
  - Lost token – election must be called
  - Failed processes – new logical ring established
Distributed Deadlock Handling

- **Prevention:** Resource-ordering deadlock-prevention
  - Define a global ordering among the system resources
  - Assign a unique number to all system resources
  - A process may request a resource with unique number \( i \) only if it is not holding a resource with a unique number grater than \( i \)
  - Simple to implement; requires little overhead

- **Prevention:** Timestamp-ordering deadlock-prevention
  - Wait-die scheme -- non-reemptive
  - Wound-wait scheme -- preemptive
  - Unique Timestamp assigned when each process is created

Prevention: Wait-Die Scheme

- Non-preemptive approach
- If \( P_i \) requests a resource currently held by \( P_j \), \( P_i \) is allowed to wait only if it has a smaller timestamp than does \( P_j \) (\( P_i \) is older than \( P_j \)). Otherwise, \( P_i \) is rolled back (release resources)

- Example: Suppose that processes \( P_1, P_2, \) and \( P_3 \) have timestamps 5, 10, and 15 respectively
  - If \( P_1 \) requests a resource held by \( P_2 \), then \( P_1 \) will wait
  - If \( P_3 \) requests a resource held by \( P_2 \), then \( P_3 \) will be rolled back
  - The older the process gets, the more waits

Prevention: Wound-Wait Scheme

- Preemptive approach, counterpart to the wait-die system
- If \( P_i \) requests a resource currently held by \( P_j \), \( P_i \) is allowed to wait only if it has a larger timestamp than does \( P_j \) (\( P_i \) is younger than \( P_j \)). Otherwise \( P_j \) is rolled back (\( P_j \) is wounded by \( P_i \))

- Example: Suppose that processes \( P_1, P_2, \) and \( P_3 \) have timestamps 5, 10, and 15 respectively
  - If \( P_1 \) requests a resource held by \( P_2 \), then the resource will be preempted from \( P_2 \) and \( P_1 \) will be rolled back
  - If \( P_3 \) requests a resource held by \( P_2 \), then \( P_3 \) will wait
  - The rolled-back process eventually gets the smallest

Deadlock Detection

- Each site keeps a local wait-for graph
  - The nodes of the graph correspond to all the processes that are currently either holding or requesting any of the resources local to that site
  - A global wait-for graph is maintained in a single coordination process: this graph is the union of all local wait-for graphs
  - There are three different options (points in time) when the wait-for graph may be constructed:
    1. Whenever a new edge is inserted or removed in one of the local wait-for graphs
    2. Periodically, when a number of changes have occurred in a wait-for graph
    3. Whenever the coordinator needs to invoke the cycle-detection algorithm
- Unnecessary rollbacks may occur as a result of false cycles
The Algorithm

1. The controller sends an initiating message to each site in the system.
2. On receiving this message, a site sends its local wait-for graph to the coordinator.
3. When the controller has received a reply from each site, it constructs a graph as follows:
   (a) The constructed graph contains a vertex for every process in the system.
   (b) The graph has an edge $P_i \rightarrow P_j$ if and only if:
       - there is an edge $P_i \rightarrow P_j$ in one of the wait-for graphs, or
   If the constructed graph contains a cycle $\Rightarrow$ deadlock.

Fully Distributed Approach

- All controllers share equally the responsibility for detecting deadlock.
- Every site constructs a wait-for graph that represents a part of the total graph.
- We add one additional node $P_{ex}$ to each local wait-for graph:
  - $P_i \rightarrow P_{ex}$ exists if $P_i$ is waiting for a data item at another site being held by any process.
- If a local wait-for graph contains a cycle that does not involve node $P_{ex}$, then the system is in a deadlock state.
- A cycle involving $P_{ex}$ implies the possibility of a deadlock.
  - To ascertain whether a deadlock does exist, a distributed deadlock-detection algorithm must be invoked.

Augmented Local Wait-For Graphs

- Site $S_1$
- Site $S_2$
- Coordinator

Augmented Local Wait-For Graph in Site $S_2$

- Site $S_2$